On the Potential of Asynchronous Pipelined Processors

Ran Ginosar and Nick Michell

UUCS-90-015

Department of Computer Science
University of Utah
Salt Lake City, UT 84112 USA

September 4, 1990

Abstract

An asynchronous version of the pipelined R3000 and DLX processors, the A3000, is being designed. Simulation was employed to investigate the potential speed-up obtainable due to the asynchronous operation. Preliminary results show up to a 64% improvement in performance.

On the Potential of Asynchronous Pipelined Processors

RAN GINOSAR* (ran@cs.utah.edu)
NICK MICHELL (michell@cs.utah.edu)

University of Utah
Dept. of Computer Science
3190 Merrill Engineering Building
Salt Lake City, Utah 84112

Abstract. An asynchronous version of the pipelined R3000 and DLX processors, the A3000, is being designed. Simulation was employed to investigate the potential speed-up obtainable due to the asynchronous operation. Preliminary results show up to a 64% improvement in performance.

1 Introduction

1.1 DLX Simulation and Software Package

The DLX (pronounced "deluxe") architecture is a simplified version of the MIPS R3000 processor [7] introduced in the textbook *Computer Architecture: A Quantitative Approach* [6,15] as an instructional tool. DLX employs the same five-stage pipeline as the R3000. The textbook is accompanied by a comprehensive software analysis set, including a compiler, a simulator, and a group of benchmark programs. Those same tools (or similar ones) have been used to generate all the measurements specified and discussed in the textbook. Thus, DLX provides a great vehicle for architectural experiments. We have applied the same architecture and tools to investigate an asynchronous processor architecture based on the DLX pipeline.

1.2 Previous Asynchronous Processor Architectures

Up until very recently, most research efforts in the area of asynchronous logic design focused at the low level of gates and very simple circuits [1,4,11,12,13,14] with some attention to the larger scope of system design [8]. Lately, however, more emphasis is being placed at the architectural level, and the first subject of investigation is naturally the general purpose processor.

The first design of an asynchronous processor was published by Martin[9]. The

*on sabbatical leave from Technion - Israel Institute of Technology, Haifa, Israel

processor was a sequential machine, based on a shared bus and a centralized control, without much parallelism. It was fabricated and tested [10], and has thus provided a solid proof of existence.

A second asynchronous processor architecture is described in [3], and is based on the design approach presented in [2]. It comprises two parallel components, a data processor and a branch processor. Both are fed by the same instruction stream. The only interaction between them is a single condition bit, notifying the latter whether or not to take a branch. While not strictly pipelined, this architecture provides for parallel execution of data operations and determination of the flow of control.

Although additional local parallelism is allowed in previous designs, it is still insufficient. To attain performance similar to or exceeding that of the best synchronous processors, a similar degree of parallelism as afforded by pipelines needs to be achieved. This is the motivation for investigating a fully pipelined asynchronous processor architecture.

1.3 Asynchronous Pipelined Architecture

Instead of inventing a new pipelined architecture, we decided to modify an existing synchronous one. Furthermore, we opted to limit the modifications to the bare minimum, so as to be able to study the effects of such marginal changes on performance. The MIPS R3000 is one of the highest performance "simple" pipelined processors. Since its simplified relative, DLX, is well documented, and a complete set of software analysis tools were available, we decided to design an asynchronous version of it, named the A3000. A full logic design and VLSI implementation are currently underway [5]. Meanwhile, we have carried out a simple experiment to investigate the potential improvement in performance, as reported below.

The synchronous pipeline of both the DLX and the R3000 consists of five stages: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), data MEMory access (MEM), and Write Back into register file (WB). In an asynchronous design, we wish to decouple the various stages, so they are separated by FIFO's. This pipeline is shown in Figure 1.

1.4 Potential of the Asynchronous Pipeline

All pipeline stages operate in lockstep mode for a synchronous processor, and each instruction usually spends one clock cycle at each stage (this preliminary study ignores floating point and some other stalls, as discussed below). All stages are very carefully balanced, so that all require approximately the same time (which is some large fraction of the clock cycle time). Each instruction has to spend a full clock cycle in each stage, even if it does not require it. For example, an ALU instruction does not require any service at the MEM stage, but it must still spend a full cycle in that stage.

Another source of waste in synchronous pipelines lies in the fact that even when a

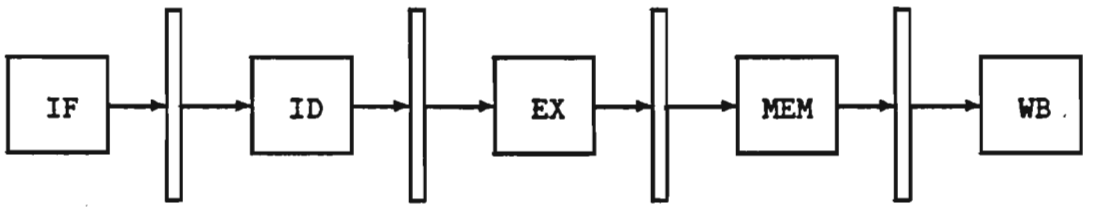

Figure 1: A3000 pipeline

pipe stage is required for an instruction, the actual computation time in that stage may be much less than a full clock cycle. For example, in the EX stage a logical operation may complete much sooner than an addition, and the addition time itself will vary depending on the carry chain length. In a synchronous design, the worst case must always be assumed for the length of the carry chain. Another example is that the ID stage, which also determines conditional branches, may take a shorter time to process non-branch instructions.

Thus, a simple asynchronous model of the A3000 would incur different processing times per each stage per each instruction. To take advantage of such flexibility, queues are provided in between the stages to accommodate waiting instructions. Note that in the simplest case a queue of depth of one is just a register. We have simulated the A3000 operation with such a model, while executing a simple benchmark, as described in the next section.

The crucial trade-off that is being made in moving from a synchronous to an asynchronous design is the addition of circuit complexity and queuing delays, while removing the constraint imposed by a global clock and allowing each stage to finish its work as soon as possible. It is clear that to realize a performance gain, the average delay in each stage must be much less than the globally worst case delay (which determines the clock period).

2 Simulation

2.1 Methodology

Several simulations are described below, but they all follow the basic description given here. A timing table is devised, as shown in part in Table 1, which specifies, for each instruction, what percentage of the full clock cycle must be spent in each pipeline stage. For example, the Add instruction spends 50% of the clock cycle in

Instruction	Equivalent Cycle Time (%) IF	ID	EX	MEM	WB
Add	50	50	50	0	50
Subtract	50	50	50	0	50
And	50	50	25	0	50
Or	50	50	25	0	50
Shift Left	50	50	50	0	50
Set Condition	50	50	50	0	50
Load	50	50	50	100	50
Store	50	50	50	100	0
Branch on Zero	50	100	0	0	0
Jump	50	100	0	0	0

Table 1: A3000 Timing Table (Selected Instructions)

each of the IF, ID, EX, and WB stages, and no time in the MEM stage. The merit and significance of this table are discussed later.

A benchmark program is executed on the existing simulator of the synchronous DLX, and the actual instruction trace is collected. The trace is then analyzed by our timing program. For each instruction in the trace, the program consults the timing table to obtain the required delays in each stage. A schedule is constructed, which runs each instruction through the pipeline, while carefully maintaining the original instruction order of the trace. Suppose that at time t instruction i completes stage p. If stage $p+1$ is empty, i.e. if instruction $i-1$ completed stage $p+1$ at time $t_2 \leq t$, then instruction i is scheduled to start at stage $p+1$ after a queuing delay (which is 10% of the cycle time for these simulations). If, on the other hand, any instruction $j < i$ still occupies stage $p+1$, instruction i is queued in the FIFO at the input to stage $p+1$.

Starting with the simplest approach, infinite queues and no dependencies, the scheduling program is gradually refined to be more realistic. The different scheduling strategies are:

level 0 infinite FIFO's, no dependencies

level 1 finite FIFO's (depth of 1 or 2), no dependencies

level 2 finite FIFO's, data dependencies

level 3 finite FIFO's, data dependencies, squashing and/or bypass

The first case assumes that queues of infinite depth are inserted between successive stages, and that there are no data or control dependencies. This case may be thought of as a loose upper bound on how much speed-up is possible. Level 1 refines this by limiting the FIFO depth to one or two, which is all that will likely be implemented (as will be argued below). Level 2 is more interesting since now data dependencies are taken into account. This is done by keeping track of busy registers. No bypass is assumed, so a register write is not completed until after WB. An instruction using a busy register as a source is stalled at ID until the register becomes free. Level 3 builds on the previous level by adding optimizations to improve performance. The two optimizations considered here are register bypass and squashing. Register bypass performs the identical function as in the synchronous design, which is to feed the result from the EX or MEM stages to the EX stage without waiting for it to go through the register file.

Squashing, as used here, has no counterpart in a synchronous design. In the asynchronous design, once an instruction completes, it can simply be dropped from the pipeline. This is used in two places in the current design. Branches complete after the ID stage; as in DLX, only simple conditions are allowed and a separate adder computes branch targets during ID. Stores complete after the MEM stage since there is nothing to write-back. Once these instructions have finished, they do not continue down the pipeline. Without squashing, completed instructions still incur queuing delays and require space in the queues. Both these effects may cause following instructions to be delayed, lowering overall performance.

One limitation of the existing DLX simulator, which affects the accuracy of the results, is that the compiler/assembler does not yet fill delay slots after branches and loads.

The simulation results depend strongly on the timing table. In this preliminary study we have only made simple assumptions regarding the values in the table, based on three observations. First, considering how much work is required per each instruction in each stage, we made some rough guesses. Second, where the computation time is data dependent, such as in some ALU operations, we chose an approximate average time, rather than the worst case time. Third, we removed external operations as bottlnecks where it seemed reasonable to do so. In the IF stage it is possible to reduce latency through faster caches or by fetching multiple instructions, so we assumed a delay of 50% of the synchronous cycle.

The timing table should be considered a design parameter, rather than fixed. Once a more detailed design has been completed and better estimates for the values for the table have been found, the simulations can be run again to obtain a more reliable performance estimate. As long as a substantial portion of the table values are significantly lower than 100% there is a gain to be made.

2.2 Performance Analysis

The simulations described above were run on a test program to quantify the potential speed-up. The test program chosen was the Dhrystone v1.1 benchmark, and a trace was generated of the first 10,000 instructions. Dhrystone is a reasonable choice because it includes a wide mix of integer instructions. The noted weaknesses of this program are not relevant here since we are not measuring compiler efficiency.

The base synchronous machine is the DLX pipeline with register bypass. Since delay slots were filled with NOPs by the compiler, no hazards will occur in the synchronous machine, which therefore executes one instruction per cycle. The asynchronous machine has a queue delay of 10% of the synchronous cycle time per stage. The results are shown in Table 2.

model	cycles	speed-up
synchronous – dependencies, bypass	10,000	1.0
infinite queues, no dependencies	5,000	2.00
1-deep queues, no dependencies	5,890	1.70
1-deep queues, data dependencies	9,150	1.09
1-deep queues, data dep., squashing	7,810	1.28
1-deep queues, data dep., bypass	7,190	1.39
1-deep queues, data dep., squashing and bypass	6,100	1.64

Table 2: Simulation Results

The table indicates that both squashing and bypass are needed to achieve a worthwhile improvement in performance over the synchronous design.

To understand the effect of queue depth, additional simulation runs were made using the model containing both bypass and squashing. Queue lengths of 0,1 and 2 were simulated under two conditions: zero queuing delay, and a delay of 10% of the synchronous cycle for each stage in the queue. A queue length of zero implies that instruction i cannot start in stage p until instruction $i-1$ has started in stage $p+1$. These results are shown in Figure 2. This clearly shows that in the case of relatively large delays per queue stage, short queues are preferred.

3 Future Plans

Three issues remain to be resolved. First, control dependencies must be considered. Second, while we have implemented a full logic design [5], we have still not resolved how to achieve asynchronous data bypasses. Third, we need to finish the logic design and its analysis before we can replace the timing table with a more realistic one.

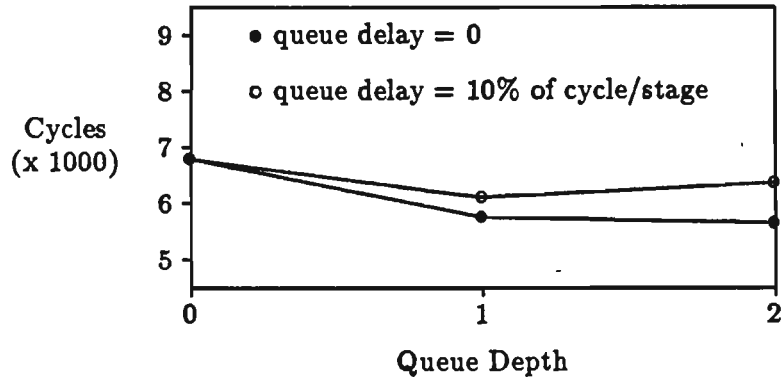

Figure 2: Effect of Queue Length on Performance

The field of computer architecture is continually moving forward, and a further challenge will be to incorporate ideas on superscalar and superpipelined design into an asynchronous environment.

4 Conclusions

We have shown that asynchronous pipelined processors can potentially outperform synchronous architectures, and have devised a method to analyze this potential.

Acknowledgements

Thanks to Erik Bruvand for his helpful comments in reviewing this manuscript.

References

1. Chu, T. *Synthesis of Self-Timed VLSI Circuits from Graph-theoretic Specifications.* PhD thesis, MIT, 1987.

2. David, I., Ginosar, R., and Yoeli, M. An efficient implementation of boolean functions and finite state machines as self-timed circuits. *Computer Architecture*

News 17, 6 (Dec., 1989), 91–104.

3. David, I., Ginosar, R., and Yoeli, M. Self-timed implementation of a reduced instruction set computer. Tech. Rep. 732, Dept. Elect. Eng., Technion, Oct., 1989.

4. Ebergen, J. *Translating Programs into Delay-Insensitive Circuits.* PhD thesis, CWI, 1989.

5. Ginosar, R., and Wolf, T. A3000: an asynchronous version of the r3000. Computer Science Department, University of Utah, in preparation.

6. Hennessy, J., and Patterson, D. *Computer Architecture: A Quantitative Approach.* Morgan Kaufman, 1990.

7. Kane, G. *MIPS RISC Architecture.* Prentice Hall, 1989.

8. Martin, A. Programming in vlsi: from communicating processes to delay-insensitive circuits. In *UT Year of Programming Inst. on Concurrent Programming*, C. Hoare, Ed., Addison-Wesley, 1989.

9. Martin, A., Burns, S., Lee, T., Borkovic, D., and Hazewindus, P. The design of an asynchronous microprocessor. Tech. Rep. CS-TR-89-02, Caltech, 1989.

10. Martin, A., Burns, S., Lee, T., Borkovic, D., and Hazewindus, P. The first asynchronous microprocessor: the test results. Tech. Rep. CS-TR-89-06, Caltech, 1989.

11. Miller, R. *Switching Theory, Vol. 2.* J. Wiley & Sons, 1965.

12. Molnar, C., Fan, T., and Rosenberger, F. Synthesis of delay-insensitive modules. *Journal of Distributed Computing* (1986), 226–234.

13. Seitz, C. System timing. In *Introduction to VLSI Systems*, C. Mead and L. Conway, Eds., Addison-Wesley, 1980, pp. 218–262.

14. Sutherland, I. Micropipelines. *Communications of the ACM* (June, 1989).

15. Wakerly, J. Book review: 'computer architecture: a quantitative approach'. *Computer Architecture News 18*, 2 (June 1990), 96–98.